Asynchronous Python for the Complete Beginner

Miguel Grinberg

S**t Programmers Say...

Async makes your code go  fast!

Wikipedia says...

“Asynchrony, in computer programming, refers to the occurrence of events independently of the main program flow and ways to deal with such events.”

How Does Python Do Many Things At Once?

- Multiple processes
  - The OS does all the multi-tasking work
  - The only option for multi-core concurrency
  - Duplicated use of system resources
- Multiple threads
  - The OS does all the multi-tasking work
  - In Python, the GIL prevents multi-core concurrency
  - Threads also consume system resources (but less than processes)
- Asynchronous programming
  - No OS intervention
  - One process, one thread
  - What’s the trick?

Real World Analogy: Chess Exhibition

Simultaneous chess exhibition by Judit Polgár, 1992 *Photo by Ed Yourdon*

- Assumptions:
  - 24 opponents
  - Polgár moves in 5 seconds
  - Opponents move in 55 seconds
  - Games average 30 move pairs
- Each game runs for 30 minutes
- 24 sequential games would take

  24 x 30 min = 720 min = 12 hours!!!

Real World Analogy: Chess Exhibition

Simultaneous chess exhibition by Judit Polgár, 1992 *Photo by Ed Yourdon*

- Polgár moves on first game
- While opponent thinks, she moves on second game, then third, fourth...
- A move on all 24 games takes her *24 x 5 sec = 120 sec = 2 min*
- After she completes the round, the first game is ready for her next move!
- 24 games are completed in *2 min x 30 = 60 min = 1 hour*

A practical definition of Async...

“A style of concurrent programming in which tasks voluntarily release the CPU during waiting periods, so that other tasks can use it.”

Async in Python

How is Async Implemented?

- Async functions must have the ability to “suspend” and “resume”
- An “event loop” keeps track of all the asynchronous functions and their stages of completion and schedules their access to CPU
- Ways to implement functions that can suspend/resume in Python:
    - Callbacks
    - Generator or coroutine functions
    - Async/await (Python 3.5+)
    - Greenlets (requires greenlet package)
- There are lots of options for asynchronous programming in Python!
    - Asyncio, Twisted, Gevent, Eventlet, Tornado, Curio, ...

Example: Standard (sync) Python

Print “hello”, wait three seconds, then print “world”

```
from time import sleep

def hello():
    print('Hello')
    sleep(3)
    print('World!')

if __name__ == '__main__':
    hello()
```

Example: Asyncio with Callbacks

Print “hello”, wait three seconds, then print “world”

```
import asyncio
loop = asyncio.get_event_loop()

def hello():
    print('Hello')
    loop.call_later(3, print_world)

def print_world():
    print('World!')

if __name__ == '__main__':
    loop.call_soon(hello)
    loop.run_forever()
```

Example: Twisted with Callbacks

Print “hello”, wait three seconds, then print “world”

```
from twisted.internet import reactor

def hello():
    print('Hello')
    reactor.callLater(3, print_world)

def print_world():
    print('World!')

if __name__ == '__main__':
    reactor.callWhenRunning(hello)
    reactor.run()
```

Example: Asyncio with Generators/Coroutines

Print “hello”, wait three seconds, then print “world”

```
import asyncio
loop = asyncio.get_event_loop()

@asyncio.coroutine
def hello():
    print('Hello')
    yield from asyncio.sleep(3)
    print('World!')

if __name__ == '__main__':
    loop.run_until_complete(hello())
```

Example: Asyncio with Async/Await (3.5+)

Print “hello”, wait three seconds, then print “world”

```
import asyncio
loop = asyncio.get_event_loop()

async def hello():
    print('Hello')
    await asyncio.sleep(3)
    print('World!')

if __name__ == '__main__':
    loop.run_until_complete(hello())
```

Example: Gevent and Eventlet

Print “hello”, wait three seconds, then print “world”

```
from gevent import sleep

def hello():
    print('Hello')
    sleep(3)
    print('World!')

if __name__ == '__main__':
    hello()
```

```
from eventlet import sleep

def hello():
    print('Hello')
    sleep(3)
    print('World!')

if __name__ == '__main__':
    hello()
```

Async Pitfalls

Pitfall #1: Async and CPU-Heavy Tasks

- Long CPU-intensive tasks must routinely release the CPU to avoid starving other tasks
- This can be done by “sleeping” periodically (such as once per loop iteration):

Asyncio:
```
await asyncio.sleep(0)
```
Twisted:
```
reactor.callWhenRunning(do_something)
```
Gevent:
```
gevent.sleep(0)
```
Eventlet:
```
eventlet.sleep(0)
```

Pitfall #2: Async and the Python Standard Library

- Blocking library functions are incompatible with async frameworks
  - socket.*, select.*
  - subprocess.*, os.waitpid
  - threading.*
  - time.sleep
- Async frameworks provide their replacements for these
- Eventlet and Gevent can “monkey-patch” the standard library
  - Enables lots of Python packages to be asynchronous!
- If nothing else works, async frameworks support running sync code in separate thread or process pools
- There is no async support for file I/O

Conclusion

Processes vs. Threads vs. Async

	Processes	Threads	Async
Optimize waiting periods	Yes (OS does it)	Yes (OS does it)	Yes
Use all CPU cores	Yes	No	No
Scalability	Low	Medium	High
Use blocking std library functions	Yes	Yes	No
GIL interference	No	Some	No

Is Async for Me?

- If you are using an async server, then you're forced to use async too
  - Examples: aiohttp, sanic, gevent, eventlet, my own python-socketio (shameless plug!)
- Writing something from scratch? The sweet spot of async is massive scaling
  - Extremely busy network servers of any kind
  - HTTP long-polling servers
  - WebSocket servers
  - Highly complex web scraping projects
  - Any other problem that requires a lot of tasks that mix network I/O with light processing
- Anything else? Probably not worth the effort
  - Unless you want to impress your friends... then go for it!

Thank You!

Miguel Grinberg
@miguelgrinberg